L27: Entry 2 of 6 File: USPT Apr 1, 2003

DOCUMENT-IDENTIFIER: US 6542855 B1
TITLE: Selecting a cache design for a computer system using a model with a seed cache to generate a trace

Brief Summary Text (22):
The major problem with the trace-generation approach is that the results are the least accurate. The model used to generate the trace shares the problem of the multiple-simulation approach that the time frame of the execution of the test program is unrealistic. The trace approach further suffers since model on which the program is executed is simpler and thus less accurate than the models (which incorporate the caches to be evaluated) used in the multiple-simulation approach.

Detailed Description Text (17):
If the trace data is stored in compressed form, it can be expanded at step S31 to provide a list of memory accesses in preparation for cache evaluation. Then, at step S32, the performance of various cache designs given the trace data is predicted so that the cache designs can be compared. The best performing cache can be selected for use in the system to be developed. Alternatively, a cost-versus-performance analysis can be conducted to determine the cache design to be selected. Note that steps S31 and S32 are run concurrently in a pipelined fashion. Step S32 can be iterated for each candidate cache design. Step S31 can be repeated for each iteration of step S32.

First Hit Fwd Refs

Generate Collection Print

L27: Entry 3 of 6 File: USPT Jan 21, 2003

DOCUMENT-IDENTIFIER: US 6510499 B1
TITLE: Method, apparatus, and article of manufacture for providing access to data stored in compressed files

Detailed Description Text (13):
FIG. 2 depicts a flow diagram 200 for compressing pages 126 of an executable file of the present invention. Typically, an operating system 120 is comprised of thousands of files 200. An executable file 202 illustratively having a size of 1 MB may be divided into 256 pages 204.sub.1 through 204.sub.256 (1,048,576 bytes/4096 bytes/page). In one embodiment of the invention, the compression buffer 140 is physically able to store 64 KB of information 206 at a time. As such, 16 pages (64 KB/ 4 KB) of the executable file 202 may be stored in the compression buffer 140 in a single instance. Once the compression buffer 140 is filled, the 16 pages of file information is compressed into a compressed block of data 208 and then stored in the permanent storage device 210 such as the flash memory. Illustratively, the compression buffer 140 performs at about a 50% compression rate. Thus, the 64 KB file is compressed to a size of 32 KB. A person skilled in the art will recognize that other compression rates, as well as various techniques, may be utilized to compress data. The compressed block 208 (32 KB at 50% compression rate) is then stored on the flash memory device 114. The foregoing compression steps are repeated for the remaining data contained in the 1 MB executable file 202. Therefore, for a 1 MB executable file, 16 blocks each comprising 16 pages of data are stored as a compressed file on the flash memory device 114.

Detailed Description Text (14):
Once the executable file 134 is compressed, the compressed data is then reorganized into a time-ordered set of data elements. In one embodiment, tracing the order in which the data is accessed during processing facilitates the reorganization. Referring to FIG. 1, a kernel trace program 132, stored on the flash memory device 114, runs in conjunction with the operating system 120 and intercepts and analyses the data from an executable file 134. Specifically, the kernel trace program 132 intercepts every interaction between a specific executable file 134 such as the "ACTLOGIN" File and the system kernel 122. The kernel trace program 132 then generates an access pattern profile for the specific executable file 134 that the kernel trace program 132 just analyzed. The access pattern profile reflects the sequence in which the pages of the opened executable file 134 are processed. Illustratively, the access pattern profile comprises a listing of the offset locations and the corresponding amount of bytes sequentially executed by the open executable file 134. In other words, the access pattern profile is a listing of each group of bytes, i.e., pages, in the order that the pages of the open file are executed by the operating system 120.

Detailed Description Text (15):
FIG. 3 depicts an illustrative access pattern profiling method 300 of tracing a file as the file is executed using the kernel trace program 132. Illustratively, the file being executed by the operating system 120 is an executable file 134 named "ACTLOGIN" (shown in FIG. 1). The method 300 begins at step 301, and proceeds to step 302 where the v-node 124 corresponding to the file ACTLOGIN file 134 is identified as the opened file that is to be traced during execution. In this instance, the v-node 124 corresponding to the open executable file ACTLOGIN 134 is

uniquely identified according to a hexadecimal address F4E3D270. In steps 304 through 324, the kernel trace program 132 generates an access pattern profile by tracing the execution of the ACTLOGIN file 134. Moreover, the access pattern profile represents a unique set of an access records, which in turn define a sequence of events during the executable file execution. Each access record comprises an offset value and size of data that is traced during the execution of the ACTLOGIN file 134.

Detailed Description Text (18):
In step 312, another 4 KB is read by the kernel trace program 132 at offset 122880. In step 414, another 4 KB is read at offset 106496. Likewise, for steps 316 through 324, 4 KB of data are sequentially read during each step. The corresponding offset values identified by the kernel trace program 132 are 110592, 118784, 114688, 77824, and 81920, which are sequentially listed as access records 7 through 11, respectively. Once each offset and byte size of the executable file 134 have been traced by the kernel trace program 132, the entire access pattern profile generated by the method 300 is stored in a temporary file (not shown) in the RAM 104 for subsequent conversion and inclusion in a relocation directory 144. The relocation directory 14 is located in the compression buffer 140. In step 326, the access pattern profiling method 300 ends. Thus, the memory management system executes pages in a non-sequential order from which the pages were initially stored. Accordingly, the access pattern profiling method 300 serves as a sequential listing of the pages as they are executed, as represented by each page's respective offset value.

Detailed Description Text (26):
The steps described above are repeated until in step 414, the compression buffer 140 has stored 16 pages, i.e., 64 KB of data. If, in step 414, the compression buffer 140 is full, then the method 400 proceeds to step 416 where the 16 pages are compressed (illustratively, into 32 KB) and then stored on the flash memory device 114. Specifically, the compressed data is stored in the order that the access records 504 were read by the kernel trace program 134 and listed with the relocation directory 144. In step 418, the 16 pages of data are discarded from the compression buffer 140 to allow for storage of additional blocks of data. The method 400 proceeds to step 404 where the compression block number 516 is set to 1 for the next 16 access records 504. Accordingly, the compression block number 516 increases by one (1) for each subsequent set of 16 optimized pages.

Detailed Description Text (32):
However, if in step 610, the requested page is not in the compression buffer 140, then the method 600 proceeds to step 614. In step 614, the compression buffer 140 will discard any previous contents and then reloads with the compressed block of data that correlates with the requested page. Thereafter, in step 614, the reloaded data is decompressed. Following the previous example, the first block (block 0) is loaded into the compression buffer 140. The method 600 then proceeds to step 612 where the 4.sup.TH page of the decompressed block of data is sent to the system RAM 104 or the device buffer for processing. The method 600 then proceeds to step 616 where a query is performed to determine if another page is demanded by the operating system 120. If in step 616, the query is affirmatively answered, then the method 600 repeats the steps 602 through 616. If, however, the query is negatively answered, then the method 600 ends in step 618.

Detailed Description Text (33):
The embodiments disclosed herein allows a memory management system of a computer system 100 to optimize demand paging in a time ordered manner. Furthermore, the pages that are optimized may be compressed in retrievable blocks of data without having to decompress all the compressed blocks within a file. As such, the computer system's processing efficiency is increased. Additionally, a data structure is generated in the form of a relocation directory. The relocation directory allows for translation between pages in a decompressed/non-optimized format and a

compressed/optimized format. Therefore, the memory management system may retrieve and execute both decompressed and compressed pages of data. Thus, a method of organizing repeatable as well as random accessed compressed and decompressed data is presented.

L27: Entry 4 of 6 File: USPT Jan 14, 2003

DOCUMENT-IDENTIFIER: US 6507921 B1
TITLE: Trace fifo management

Brief Summary Text (5):
Microprocessors are general purpose processors which require high instruction throughputs in order to execute software running thereon, and can have a wide range of processing requirements depending on the particular software applications involved. A software developer may want to trace the execution sequence of a program in order to determine actual execution sequence and then modify the program in order to optimize execution performance. Similarly, a software developer may want to trace the execution sequence of a program in order to identify an error. However, tracing a processor with limited external buses or on board caches is difficult or impossible.

Detailed Description Text (101):
In the case in which a call instruction inside a repeat block causes another repeat block to be executed then this is considered as level 2 nesting. Table 24 illustrates a typical case. In this case, even if TRC_RPT=0 only RPTB2 is compressed. RPTB1 will be traced fully for all iterations.

CLAIMS:

8. A method of operating a digital system comprising a microprocessor, wherein the microprocessor is operable to trace a sequence of instruction addresses, comprising the steps of: providing an instruction address that identifies a first instruction in a sequence of instructions to be decoded by an instruction buffer unit; decoding the first instruction of the sequence of instructions in the instruction buffer unit; tracing the instruction address of the first instruction by storing the address of the first instruction only if the first instruction is adjacent to a discontinuity in the sequence of instruction addresses; repeating the steps of providing, decoding and tracing to form a sequence of discontinuity addresses; and wherein the step of tracing further comprises storing a compressed representation of a sequence of instructions executed in a linear manner.

13. A method of operating a digital system comprising a microprocessor, wherein the microprocessor is operable to trace a sequence of instruction addresses, comprising the steps of: providing an instruction address that identifies a first instruction in a sequence of instructions to be decoded by an instruction buffer unit; decoding the first instruction of the sequence of instructions in the instruction buffer unit; tracing the instruction address of the first instruction by storing the address of the first instruction only if the first instruction is adjacent to a discontinuity in the sequence of instruction addresses; repeating the steps of providing, decoding and tracing to form a sequence of discontinuity addresses; wherein the step of tracing further comprises: storing a first length format data item indicative of a length of the first instruction to form a sequence of instruction lengths; storing a compressed representation of a sequence of instructions executed in a linear manner; storing a first discontinuity event type data item if the first instruction is adjacent to a discontinuity in the sequence of instructions, whereby the cause of the first discontinuity is indicated; and storing the instruction address of the first instruction only once if the first

instruction is a repeat instruction.

L27: Entry 5 of 6 File: USPT Feb 12, 2002

DOCUMENT-IDENTIFIER: US 6347383 B1
TITLE: Method and system for address trace compression through loop detection and reduction

Detailed Description Text (4):
Referring throughout to FIG. 6, there is depicted a flowchart showing the overall sequence of taking and compressing a trace using the aforementioned tracing tool used in association with the further compression technique of the present invention. First though, it follows that traditional tracing mechanisms based on program instrumentation would not succeed in generating the required information, because such instrumentation only deals with effective addresses, not virtual addresses. Therefore, kernel level access is essential to be able to read the segment registers and record them in the trace. Furthermore, the large 56-bit virtual addresses puts more pressure on the trace buffer and generates larger file sizes than other architectures. The tracing tool utilized in association with the present invention depends on a combination of hardware assist and simple kernel level instrumentation to capture memory references during a trace. The hardware assist consists of special registers in the PowerPC processor architecture that force a processor interrupt when specific events occur. The software instrumentation is in the kernel and consists of an interrupt handling routine that takes over whenever the hardware assist forces an interrupt. To generate a trace, the registers are set to interrupt the processor whenever an instruction generates a load or store, a branch instruction executes (conditional or otherwise), or an interrupt occurs that interrupts the sequential flow of the program (hardware interrupts or software signals, for instance). In any of these cases, the operating system takes over and the interrupt handling routine generates a trace record containing the 32-bit effective address of the instruction, in addition to the 56-bit virtual address of the data being loaded or stored, if applicable.

Detailed Description Text (8):
Next, the method of the present invention identifies loops within the program structure using standard control flow analysis in step 64 resulting in the trace record shown in FIG. 4. Informally, a loop construct is defined as a sequence of basic blocks such that there is only one entry to the sequence from outside, and there are backward branches to that entry from within the sequence. This is a conventional definition that has been used in the prior art in program optimization in the compiler area. Loops may be nested, and such nesting detected by traditional control flow techniques adapted to read and analyze the trace instruction flow. Therefore, for the purpose of compressing the trace, there are identified three types of load and stores within a loop. Referring to FIG. 4, the first type of loop are constant addresses 40. These do not change from one loop iteration to the next. This occurs for example when a stack variable is repeatedly read into a register (spill code), or some similar situation. The second are offset or loop-variant addresses 42. These addresses 42 change from one iteration of the loop to the next by a fixed offset. The third is chaotic, random or variable addresses 44. These addresses 44 change from one iteration of the loop to the next without following any clear pattern.

Refine Search

Your wildcard search against 10000 terms has yielded the results below.
Your result set for the last L# is incomplete.
The probable cause is use of unlimited truncation. Revise your search strategy to use limited truncation.

Search Results -

Terms	Documents
(h$ near5 flow$ near4 pipelin$)	1

Database:
US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L33

Refine Search | Recall Text | Clear | Interrupt

Search History

DATE: Friday, June 18, 2004 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=EPAB; PLUR=YES; OP=ADJ			
L33	(h$ near5 flow$ near4 pipelin$)	1	L33
DB=USOC; PLUR=YES; OP=ADJ			
L32	(h$ near5 flow$ near4 pipelin$)	89	L32
DB=PGPB; PLUR=YES; OP=ADJ			
L31	(h$ near5 flow$ near4 pipelin$)	4	L31
L30	(h$ near5 flow$ near4 pipelin$)	4	L30
DB=USPT; PLUR=YES; OP=ADJ			
L29	(h$ near5 flow$ near4 pipelin$)	18	L29
L28	(h-flow near4 pipelin$)	0	L28
L27	L26 and compress$ same (recurrent$ or repeat$)	6	L27

L26	(creat$ or generat$ or mak$ or develop$) near9 trace$ near9 execut$	316	L26
L25	L24 and compress$	6	L25
L24	L22 and execut$	33	L24
L23	L22 and (execut$ near9 (traci$ or trace$))	0	L23
L22	recurrent$ and (reus$ or re-use$ or (re$ near4 us$)) near9 comput$	71	L22
L21	L20 and (instanc$ near9 instruction$)	11	L21
L20	L19 and (phras$ or text$)	135	L20
L19	L18 and map$ and instruction$ and (processor$ or execut$)	263	L19
L18	(reus$ near9 comput$)	880	L18
L17	(recurrent$ near9 reus$)	4	L17
L16	L13 and (map$ and instruction$ and processor$) and reusabl$	3	L16
L15	L13 and (map$ and instruction$ and processor$) and recurrent$ and reusabl$	0	L15
L14	L13 and (map$ near7 instruction$)	12	L14
L13	(n$ near4 dimension$) near8 (one$ near4 dimension$)	1088	L13
L12	(n -dimension$) near8 (one$ near4 dimension$)	0	L12
L11	L9 and l1	3	L11
L10	L9 and l2	0	L10
L9	717/127,128,131.ccls.	510	L9
L8	L7 and l5	0	L8
L7	717/127,128,131.ccls.	510	L7
L6	L5 and (instanc$ near8 (process$ or instruction$))	2	L6
L5	L2 and (processor$ or execut$) near5 instruction$ and (instanc$)	31	L5
L4	L2 and (processor$ or execut$) near5 instruction$ near9 (instanc$)	0	L4
L3	L2 and (processor$ or execut$) near5 instruction$	31	L3
L2	L1 and (state$ near4 vector$)	45	L2
L1	(many$ near6 one$) near6 map$	689	L1

END OF SEARCH HISTORY

Refine Search

Search Results -

Terms	Documents
L2 and (processor$ or execut$) near5 instruction$ and (instanc$)	31

Database:
US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L5

Refine Search | Recall Text | Clear | Interrupt

Search History

DATE: Friday, June 18, 2004 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L5	L2 and (processor$ or execut$) near5 instruction$ and (instanc$)	31	L5
L4	L2 and (processor$ or execut$) near5 instruction$ near9 (instanc$)	0	L4
L3	L2 and (processor$ or execut$) near5 instruction$	31	L3
L2	L1 and (state$ near4 vector$)	45	L2
L1	(many$ near6 one$) near6 map$	689	L1

END OF SEARCH HISTORY

Terms used
h flow and **pipeline** and **execute** and **execution** and **processor** and **instruction**

Found **41,507** of **138,517**

Sort results by: relevance

Display results: expanded form

Save results to a Binder

Search Tips

Open results in a new window

Try an Advanced Search

Try this search in The ACM Guide

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next

Best 200 shown

Relevance scale

1 Instruction fetch mechanisms for multipath execution processors

Artur Klauser, Dirk Grunwald

November 1999 **Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture**

Full text available: pdf(1.43 MB) Publisher Site

Additional Information: full citation, abstract, references, citings, index terms

Branch mispredictions can have a major performance impact on high-performance processors. Multipath execution has recently been introduced to help limit the misprediction penalties incurred by branches that are difficult to predict. This paper presents efficient instruction fetch architecture designs for these multipath processor execution cores. We evaluate a number of design trade-offs for the first-level instruction cache and the multipath PC fetch arbiter. Furthermore we evaluate the e ...

2 An evaluation of speculative instruction execution on simultaneous multithreaded processors

Steven Swanson, Luke K. McDowell, Michael M. Swift, Susan J. Eggers, Henry M. Levy

August 2003 **ACM Transactions on Computer Systems (TOCS)**, Volume 21 Issue 3

Full text available: pdf(578.85 KB) Additional Information: full citation, abstract, references, index terms

Modern superscalar processors rely heavily on speculative execution for performance. For example, our measurements show that on a 6-issue superscalar, 93% of committed instructions for SPECINT95 are speculative. Without speculation, processor resources on such machines would be largely idle. In contrast to superscalars, simultaneous multithreaded (SMT) processors achieve high resource utilization by issuing instructions from multiple threads every cycle. An SMT processor thus has two mean ...

Keywords: Instruction-level parallelism, multiprocessors, multithreading, simultaneous multithreading, speculation, thread-level parallelism

3 SIMP (Single Instruction stream/Multiple instruction Pipelining): a novel high-speed single-processor architecture

K. Murakami, N. Irie, S. Tomita

April 1989 **ACM SIGARCH Computer Architecture News , Proceedings of the 16th annual international symposium on Computer architecture**, Volume 17 Issue 3